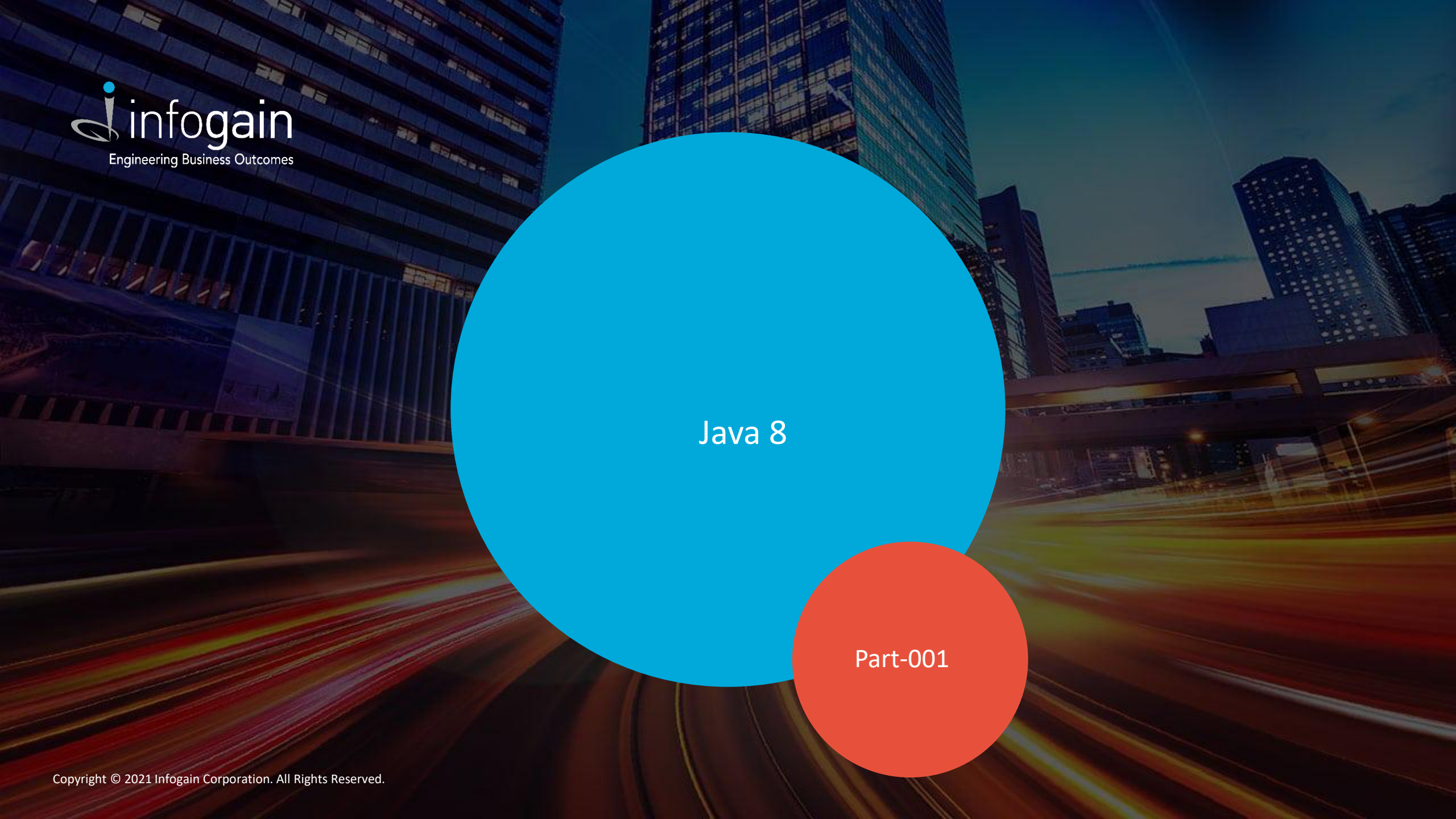

Objectives

- Able to explain and use default and static method
- Able to explain Functional Programming
- Abe to explain and use Functional Interface
- Able to explain and write lambda expressions
- Able to use in-built functional interfaces of **java.util.function** package
- Able to use method and constructor reference
- Able to process data from the Collection API

Default Method

- A default method is a method which is declared using “default” keyword and it contains a method body.
- Default method is available to all implementing classes of that interface.
- Implementing class may/may not override the default method.
- Default methods are introduced to add extra features to current interfaces without disrupting their existing implementations.
- All method declarations in an interface, including default methods, are implicitly public, so you can omit the public modifier.

Why Default Method?

- **Answer**
  - Default method helps with backward compatibility without doing reengineering an existing JDK framework.
- **Let's Understand By Example**
- For example, **stream()** is a default method which is added to Collection interface in Java 8.

```
default  Stream <E>  stream(){
return StreamSupport.stream(spliterator(),false);
}
```

- The stream method is required in all **List** and **Set** implementations so added in their super interface i.e., **Collection**. Doing this, stream method will now be directly available to all their implementing classes **ArrayList**, **TreeSet**.
- If stream() would have been added as abstract method, then all classes implementing Collection interface needs to implement stream() method which requires lots of Reengineering.
- Thanks to Java 8 default method feature, now it is a default method, all implementations of **Collection** interface inherit default implementation of **stream()** method.

Default Methods: Rules

- Default methods are public implicitly, just as any other method of an interface
- An interface can have any number of abstract and default methods
- All methods with the keyword default must have a body
- Default methods cannot be final
- Default methods cannot be synchronized.
- You cannot use Object class method as default method.

Default Methods: Rules(contd..)

- **Let's Understand Rules by Example**

```
public interface Person {
 default  final void sayHello1() {   // C.E      default method can not be final
 System.out.println("Welcome to the World!!");
 }
 default synchronized void sayHello2(){  // C.E      default method can not be synchronized
        System.out.println("Welcome to the World!!");
 }
 default String toString(){  // C.E Object class method can not be used as Default method
 {}
```

Default Method and Multiple Inheritance

- By introducing default methods, it opened the gate for **diamond problem** where a class can inherit two methods of the same signature from two different interface/classes.
- As from Java 8 interfaces can contain concrete methods, there is the possibility of a class inheriting more than one method with the same signature.
- **Java 8 acknowledges this conflict with three basic principles.**
  - A method declaration in the class or a superclass wins the priority over any default method declared in the interface.
  - The method with the same signature in the most specific default-providing interface will take the priority.
  - In case choices are still ambiguous, the class inheriting from multiple interfaces has to override the default method and then it can provide its own implementation. To call the super interface method super keyword is used.

Static Methods

- **Static Method**
  - Static methods are public implicitly, just as any other method of an interface
  - Static methods in interfaces are defined just like static methods in classes, with the keyword static
  - Static method cannot be overridden, and an interface can have any number of static methods
- **Why Static Methods?**
  - You know Collection and Collections.
  - Collection is an interface and Collections is a utility class containing only static methods which operate on Collection objects.
  - Java API developers have followed this pattern of supplying a utility class along with an interface to perform basic operations on such objects.
  - But from Java 8, they have broken this pattern by introducing static methods to interfaces.
  - With the introduction of static methods to interface, such utility classes will disappear and methods to perform basic operations will be kept as static methods in interface itself.
- **Example of Interfaces containing static methods**
  - Stream.of, Comparator.naturalOrder, Comparator.comparing etc

What is functional Programming?

- Functional programming is a programming paradigm, which concentrates on computing results rather than on performing actions.
- Functional Programming is based on lambda calculus and avoids changing-state .
- Functional Programming supports parallel data processing
- Functional programming is more declarative rather than imperative.
- Declarative programming approach seeks to describe what you want to do instead of specifying how you want to do.

Example of Imperative Approach

- Let’s get clear on this : Suppose you have a list of employees and you just want to fetch the name of all those employee whose salary is greater than x amount in a imperative way?

```
private List<String> filterEmployee(List<Employee> employees) {
List<String> names = new ArrayList<String>();
for(Employee emp: employees){
if(emp.getSal() > 30000){
names.add(emp.getName());
} }
return names; }
```

- **Here, we're saying:**
  - Create a result collection
  - Step through each Employee in the collection
  - Check the salary, if it's greater than 30000, add it to the results

Example of Declarative Approach

- With declarative programming, on the other hand, you write code that describes what you want, but not necessarily how to get it (declare your desired results, but not the step-by-step):

```
private List<String> filterEmp(List<Employee> employees) {
List<String> names = new ArrayList<String>();
names=  employees.stream().filter(emp->emp.getSal()>30000).map(emp->emp.getName()).collect(Collectors.toList());
return names;
}
```

- Here, we're saying give us Employee name where Employee salary is greater than 30000 , we are not iterating the collection using loop no if statement is used and able to implement same functionality with less amount of code .

Functional Interface

- **Functional Interface**
  - Any interface having exactly one abstract method inside it is called functional interface. It is also called SAM(Single Abstract Method) interface.
  - Functional interfaces are useful for lambda expression because the signature of the abstract method describe the signature of a lambda expression's parameter and return type.
  - Functional interface provides target types for lambda expressions and method references.
- **Example**

```
pubilc interface Runnable{
public void run();
 }
```

Functional Interface(contd..)

- **Functional Interface Rules**
  - A functional interface is any interface that has exactly one abstract method.
  - Functional interface can have any number default methods .
  - If an interface declares an abstract method with the signature of one of the methods of **java.lang.Object**, it doesn't count toward the functional interface method count.
  - An empty interface is not considered a functional interface.
  - A functional interface is valid even if the **@FunctionalInterface** annotation would be omitted.
- **Below Code Snippet is an Example of Functional Interface.**

```
@FunctionalInterface
public interface MyFunctionalInterface {
   public abstract void execute();
   @Override
   String toString();
   default void beforeTask()  {       Code Here     }
   default void afterTask() { Code Here }
}
```

What is a lambda expression?

- The term lambda expression comes from lambda calculus.
- In programming Lambda expression is an anonymous function. A function that doesn’t have a name and doesn’t belong to any class.
- A lambda expression has list of parameters, function body and returns result.
- It is used to implement a method defined by a functional interface .
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.

Parts of Lambda Expression

- **Basically, Lambda has 3 parts.**
  - **A list of parameters:** Comes before arrow symbol
  - **Function body :** The behavior comes after arrow
  - **An arrow :** Separator between parameter list and function body
- **Syntax of Lambda Expression**

  (parameter_list) -> {function_body }
- **Example**
  - (x, y) -> x + y This lambda expression takes two arguments x and y and returns the sum of these two arguments.

Parts of Lambda Expression(contd..)

- **The Parameters**
  - A lambda expression can receive zero, one or more parameters.
  - The type of the parameters can be explicitly declared, or it can be inferred from the context.
  - Parameters are enclosed in parentheses and separated by commas.
  - Empty parentheses are used to represent an empty set of parameters.
  - When there is a single parameter, its type is inferred, and it is not mandatory to use parentheses.
- **Example**
  - `() -> System.out.println("Hi");`
  - `(String s) -> System.out.println(s);`
  - `(s) -> System.out.println(s);` The type of the parameters can be declared explicitly, or it can be inferred from the context:
  - `s -> System.out.println(s);` If there is a single parameter, the type is inferred, and it is not mandatory to use parentheses
  - `(String s1, String s2) -> System.out.println(s1 + s2);`

Parts of Lambda Expression(contd..)

- **An arrow**
  - Formed by the characters - and > to separate the parameters and the body.
- **A body**
  - The body of the lambda expressions can contain one or more statements.

```
(int a) -> a*6;
```

- If lambda body enclosed inside curly braces, then return keyword is required in case your behavior returns value

```
(int a) -> {  System.out.println(a);  return a*6; }
```

- If the lambda expression doesn’t return a result, a return statement is optional

```
() -> System.out.println("Hi");
() -> {
System.out.println("Hi");
return
}
```

So, What Is a Java 8 Lambda Expression?

- Answer: To make instances of anonymous classes easier to write and read!
- Live coding : Runnable, Comparator
- **Three Questions About Lambdas**
- **What is the type of a lambda expression?**
  - Answer: a functional interface
- **Can a lambda be put in a variable?**
  - **Answer is Yes!**
  - Comparator<String> c= (String s1,String s21)-> Integer.compare(s1.length(),s2.length());
- **Consequences** : Lambda can be passed as method parameter, and can be returned by the method

Is a Lambda an Object?

- **Let's compare the following**

```
Comparator<String> c=(String s1,String s21)->
 Integer.compare(s1.length(),s2.length());
Comparator<String> c=new Comparator<String>(String s1,String s2){
boolean public compareTo(String s1,String s2){
Integer.comapre(s1.length(),s2.length());
}
};
```

- A lambda expression is created without using <<new>> keyword
- The answer is complex, but no
- Exact answer: a lambda is an object without an identity

Java In-Built Lambda Interfaces

The **java.util.function** package contains rich set of functional interfaces used by JDK and also available for end users.

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type **T** and return the result, which is also of type **T**. Its method is called **apply**( ).
BinaryOperator<T>	Apply an operation to two objects of type **T** and return the result, which is also of type **T**. Its method is called **apply**( ).
Consumer<T>	Apply an operation on an object of type **T**. Its method is called **accept**( ).
Supplier<T>	Return an object of type **T**. Its method is called **get**( ).
Function<T, R>	Apply an operation to an object of type **T** and return the result as an object of type **R**. Its method is called **apply**( ).
Predicate<T>	Determine if an object of type **T** fulfills some constraint. Return a **boolean** value that indicates the outcome. Its method is called **test**( ).

Consumer Interface

- **java.util.function. Consumer<T>**
  - A consumer is an operation that accepts a single input argument and returns no result .it just execute some operations on the argument provided to it.
- **Interface Definition**

```
public interface Consumer<T> {
void accept(T t);
}
```

Predicate Interface

- **java.util.function.Predicate<T>**
  - A predicate is a statement that may be true or false depending on certain condition.
  - **java.util.function.Predicate** has a boolean-valued function that takes an argument and returns boolean value.
  - This functional interface can be used anywhere you need to evaluate a boolean condition.
- **Interface Definition**

```
public interface Predicate<T> {
boolean test(T t);
}
```

- **Example**

```
List<String> myNames=Arrays.asList("Amit","Anuj","Äbha",“Java")
Predicate<String> p2=t -> t.endsWith("a");
```

Function Interface

- **Function<T, R>**
  - A function represents an operation that takes an input argument of a certain type and produces a result of another type.
  - A common use of Function interface is to convert or transform from one object to another
- **Interface Definition**

```
public interface Function<T, R> {
R apply(T t);
}
```

Supplier Interface

- **java.util.function. Supplier<T>**
  - Supplier is a functional interface which does the opposite of a consumer it takes no arguments and returns a result.
  - Supplier can be used in all contexts where there is no input but an output is expected.
  - Since Supplier is a functional interface, hence it can be used as the assignment target for a lambda expression or a method reference and constructor reference.
  - It represents a function which does not take in any argument but produces a value of type T.
  - The lambda expression assigned to instance of Supplier type is used to define its get() which eventually produces a value.
- **Interface Définition**

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

Supplier Interface(contd..)

Below Supplier instance return a Random Number

```
// This function returns a random value.
Supplier<Double> randomValue = () -> Math.random();
// Print the random value using get()
System.out.println(randomValue.get());
```

Method reference

- **Method Reference**
  - Method reference is used to refer method of functional interface. It represent instance of a functional interface.
  - A method reference is a simplified form (or short-hand) of a lambda expression that executes just one method.
  - Instead of using an anonymous class you can use a Lambda expression and if this just calls one method, you can use a method reference
  - Method references can refer to both static and instance methods of class, and it will enhance the readability of your code.

- **Syntax of Method Reference**
  - **CLASS OR INSTANCE THAT CONTAINS THE METHOD :: METHOD NAME TO EXECUTE**
  - :: Double colon is method reference operator

Method Reference(contd..)

- **Static Method Reference Example**

  Function<Integer ,Double> sqrt= (n)->Math.sqrt(n);

  Function<Integer ,Double> sqrt= Math::sqrt;

- **Instance Method Reference Example**

  Function<String ,String> lowerCaseFunction= (s)->s.toLowerCase();

  Function<String ,String> lowerCaseFunction= String::toLowerCase;

Method Reference(contd..)

- **Types Of Method Reference**
  - Reference to a static method of class.
  - Reference to an instance method of class.
  - Method reference to instance method of non-existing object
  - Reference to a constructor of a class.
- **Method reference to static method :**
  - ClassName::MethodName
  - Use this syntax when you are referring to a static method.
- **Example:**
  - Integer::parseInt, Math::max

Method Reference(contd..)

- **Method reference to instance method of an existing object :**
  - **ReferenceVariable::MethodName**
  - Use this syntax when you are referring to an instance method of already existing object.
- **Example :**
  - **e::getName**
  - here ‘e’ is a reference variable referring to Employee object which already exist.
- **Method reference to instance method of non-existing object :**
  - **ClassName::MethodName**
  - Use this syntax when you are referring to an instance method by passing reference variables as an argument.
- **Example :**

  (Employee e) -> e.getName() can be written as Employee::getName

Method Reference(contd..)

- **Constructor References**
  - You can also refer to the constructor of a class same as method references.
  - Syntax for referring to constructor is, **ClassName::new**
- **Example :**
  - Employee::new

So, What Do We Have so Far?

- A new concept: the « lambda expression », with a new syntax
- A new interface concept: the « functional interface »
- Question: how can we use this to process data?

How Do We Process Data in Java?

- **Where are our objects ?**
  - Most of the time : In collection (may be a List , Set or Map)
  - Can we process this data using lambdas?
- The good new is yes!

```
List<Customer> list=...;
list.forEach(customer ->  System.out.println(customer));
```

- **Or**

```
List<Customer> list=...;
list.forEach(System.out::println);
```

- **But… where does this forEach() method come from?**
- Adding a forEach() method on the Collection interface breaks the compatibility : all the implementations must be refactored!

How to Add Methods to Iterable?

```
public interface Iterable<E>{
void forEach(Consumer<E> consumer);
 }
```

- **Above declaration break all the existing implementations ,refactoring these implementations is not an option**
- **If we put the implementation ,then it will not break the existing implementations?**

```
 public interface Iterable<E>{
default void forEach(Consumer<E> consumer) {
 for(E e:this){
 consumer.accept(e)     }
  }   }
```

infogain
Engineering Business Outcomes

Contact :

Infogain Corporation, HQ

485 Alberto Way
Suite 100
Los Gatos, CA 95032

Phone: 408-355-6000

Irvine

41 Corporate Park
Suite 390
Irvine, CA 92606

Phone: 949-223-5100

Austin

111 W. Anderson Lane
Suite E336
Austin, TX 78752

Phone: 512-212-4070

Phoenix

21410 North 19th Ave
Suite 114
Phoenix, AZ 85027

Phone: 602-455-1860

London

Millbank Tower, Citibase 21-24
Millbank, office no 1.7
London SW1P 4DP

Phone: +44 (0)20 3355 7594

Noida

A-16 & 21, Sector 60
Gautam Budh Nagar
Noida – 201 301, India

Phone: +91 12 0244 5144

Mumbai

Unit No. 74, 2nd Floor, SDF 3
SEEPZ, Andheri East
Mumbai – 400 096, India

Phone: +91 22 6114 6969

Bengaluru

#7, 18th Main Road, 7th Block
Koramangala
Bengaluru - 560 095, India

Phone: +91 80 4110 4560

Dubai

Office No. 255, Building No. 17
Dubai Internet City
Dubai, UAE

Phone: +971-4-458-7336

Singapore

144 Robinson Road, #13-01
Robinson Square
Singapore 068908

Phone: +65 62741455